Graph Theoretic Approach on Branch Change

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Vishnu Teja
(111601028)

under the guidance of

Dr. Deepak Rajendraprasad

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Graph Theoretic Approach on Branch Change**" is a bonafide work of **Vishnu Teja** (**Roll No. 111601028**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

Dr. Deepak Rajendraprasad

Assistant Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Contents

Chapter 1

Introduction

After the end of the first semester in B.Tech curriculum, every student has an option to request for a change of their department. Depending on the candidate's performance in the first semester, he/she can get their desired branch if this particular transition dosen't violate certain set of rules declared by Institute Senate. This is called as the Branch Allocation problem.

Since the establishment of our Institute, our academic section has been solving this problem manually. The chances of error and time consumption is high when it is done manually. So, this project aims to deliver a computable program to allocate the students who requested branch change to their highest Possible preference branches without violating the rules.

1.1 Organization of the Report

This chapter vaguely describes the problem and the need to solve this. The rest of the chapters are organised as follows: next chapter we understand the problem in detail and in chapter 3 we provide review of prior works. In Chapter 4 we discuss the algorithm and the theoretical proof[4.4]. In Chapter 5 we see the Implementation of this algorithm and

a web interface[5.3] for the same. In Chapter 6 we discuss the attempts made to solve the problem with Rule-5. And finally in Chapter 7, we conclude with some future works.

Chapter 2

Understanding the Problem

2.1 Rules and Ordinances

According to the rules and ordinances declared by the Senate of Indian Institute of Technology Palakkad, change of branch may be permitted subject to the following rules:

1. Such change will be considered only at the end of the first semester. The performance during the first semester will be the basis for consideration for change of branch.

2. All students who have successfully completed the first semester of the course, except Engineering Design Dual Degree students, will be eligible for consideration for change of branch, subject to the availability of vacancies.

3. In making a change of branch, the strength of a class should not fall below the existing strength by more than ten percent and should not go above the sanctioned strength by more than ten percent. For this purpose, the strength in both cases refers to the total strength of the students in the class.

4. However, a minimum of one student will be eligible for consideration for change of branch from each discipline at the end of the first semester, irrespective of the third regulation.

5. If a student with a higher GPA is not offered a particular branch because of other constraints, this should not be offered to any other students with a lower GPA even if he/she is eligible on the basis of the existing norms.

6. Change of branch rules is subject to revision from time to time and the decision of the Senate will be final and binding.

2.1.1 Sanctioned Strength

The number of seats alloted to a particular branch is the Sanctioned strength. Currently, our Institute has 4 branches and their sanctioned strengths are as follows,

Computer Science and Engineering (CSE) - 50,

Electrical Engineering (EE) - 50,

Mechanical Engineering (ME) - 30,

Civil Engineering (CE) - 30

2.1.2 Existing Strength

The number of students who have joined a particular branch at the start of the first semester is called Existing Strength. Existing Strength may not be equal to Sanctioned Strength as some of the seats may be vacant.

- Rule 3 states that the strength of a class should not fall below the existing strength by more than 10 percent. For example, say for CSE the existing strength is 48. 90 percent of 48 = 43.2. Which means the strength can fall down upto 44 only, because falling down to 43 will violate the rule.But in practice, the strength is allowed to fall by the smallest integer bigger than 10 percent of existing strength and hence if the existing strength is 48, the new strength can be 43 but not lower.We call this situation as *Floored out* or *Bottomed out*.

- Rule 3 states that the strength of a class should not go above the sanctioned strength

by more than 10 percent. For example, say for CSE the sanctioned strength is 50. 110 percent of 50 = 55. Which means the strength can go to atmost 55.We call this situation as *Roofed out* or *Maxed out*.

- Rule 4 is a sub-case of Rule 3. The word irrespective of rule 3 is used to tackle situations like these.. For example, a branch existing strength is only 8 (any natural number less than 10 can be taken as example) in this case not even one student can leave the branch because if one student leaves then the strength falls below 90 % of existing strength. So, in such cases the Rule 4 ensures that one student can go out of that branch.

2.1.3 Origin of Rule - 5

The origin of Rule - 5 probably would have come up to ensure that this reallocation is done in a fair manner. Though at a first look, this rule seems fair for a higher GPA student, but this rule can also adversely affect a higher GPA student.

2.1.4 Rules of some other IITs

Institute	L-Limit [1]	U-Limit[2]	Minimum CGPA	Rule-5	Reference
IIT Kanpur	60% of SS[3]	Greater of SS or ES[4]	Nil	No	IITK [6]
IIT Kharagpur	Nil	110% of SS	≥ 8.5	No	IITKGP [7]
IIT Madras	90% of ES[4]	110 % of SS	Nil	Yes	IITM [8]
IIT Bombay	75% of ES	110 % of SS	Nil	Yes	IITB [9]
IIT Guwahiti	90% of ES	110 % of SS	Nil	No	IITG [10]
IIT Roorkee	Nil	110 % of SS	Nil	No	IITR [11]
IIT Bhuvaneshwar	90% of ES	110 % of SS	≥ 8.5	No	IITBBS [12]
IIT Jodhpur	95% of ES	115 % of SS	≥ 8	No	IITJ [13]
IIT Tirupathi	90% of ES	110 % of SS	Nil	Yes	IITT [14]
IIT Palakkad	90% of ES	110 % of SS	Nil	Yes	IITPKD [2]

2.1.5 Supremum Criteria for Allocations

This criteria gives more priority to the student with Highest GPA. Suppose let us say there are two allocations A_1 and A_2 that are sorted in descending order of GPAs. We say A_1 is optimal / better than A_2 if student S gets a better preference in A_1 compared to A_2. Where S is the first student where the allocations differ from each other from the top.

[1] Lower Limit for a branch

[2] Upper limit for a branch

[3] Sanctioned strength of a branch

[4] Existing Strength of a Branch

Chapter 3

Review of Prior Works

Last year my senior K.Durga Prasad worked on this problem and made some important observations and has come up with an algorithm to tackle this problem.

3.1 Algorithm

With a given set of students and their branch preferences,there could be multiple possible allocations.We can consider the allocation where more no.of students get their preferred branch or the allocation where the highest preference is given to the highest CGPA student no matter what the other lower CGPA students get. We consider the **Supremum** allocation [2.1.5] as optimal allocation i.e the allocation in which highest priority is given to the student with highest CGPA. The algorithm is as follows:

1. List the students in the descending order of their GPAs. In case of a GPA clash, the student with better JEE rank will be ranked ahead.

2. Create a matrix named "outputMatrix" with dimensions (no. of students * no. of branches). Each row represents a student in our ranked order.

3. Each column represents a Preference branch. 1st column represents Preference branch 1, 2nd column represents preference branch 2, 3rd column represents preference

branch 3, and so on and the last column represents his own branch.

4. Every value in the matrix is either 1 or 0.1s in the matrix represent the branch a particular student got into, 0s in the matrix represent that the student didn't get into that particular branch.

5. Initialize the matrix by giving every student his 1st Preference. This is the best allocation possible. Here best in the sense every student gets their best possible preference.

6. Check whether any of the rules are violated? The Strength Rule which is Rule - 3 and Rule - 5.If the rules are not violated then this is the best possible allocation.

7. If the allocation violates the rules then check for the next possible best allocation i.e [1,1,1,1....1,2] and then [1,1,1,1,...,1,3] and next [1,1,1,1,...,2,1] and so on..

8. This continues until an allocation satisfies all the rules in the Rulebook.

3.2 Conclusion

This algorithm gives the best possible solution for a given set of requests for branch change considering every possible allocation.But Running time of the algorithm is exponential. Let b be the no of branches and s be the no of students. It is in the order of b^s.

Chapter 4

Algorithm

We model this problem as finding an Eulerian digraph S in a digraph D, so that S represents the optimal solution as per supremum criteria. An Eulerian digraph is a digraph in which the in-degree and out-degree of every node are equal. In section 4.1 we discuss the construction of the Graph. In section 4.2 we present the Definitions and Observations used in the algorithm. In section 4.3 we discuss the algorithm and in section 4.4 we present the Proof of Correctness of the Algorithm.

4.1 Graph Construction

1. Let us call the directed graph we are constructing as Request Graph (D).
2. Create a node for every student who applied for branch change and every Branch available.
3. Also create a node called Buffer to accommodate the variable strengths of the Branches.
4. V(D) = $Students \cup Streams \cup \{Buffer\}$
5. Let the number of students be s, the number of branches be b. So, the graph D has $s + b + 1$ vertices.

6. For each student, add an arc of weight b from his current branch to him.

7. Add arcs from students to their preferred branches with preference numbers as weights.

8. Add x arcs from buffer to a branch where x is the max reduction allowed for that branch (i.e., current strength - minimum allowed strength)

9. Add y arcs from branch to Buffer where y is the maximum increase allowed (i.e. maximum allowed strength - current strength)

See Fig : 5.1 in the next chapter for example.

4.2 Definitions and Observations

1. **Eulerian digraph:** A digraph D is Eulerian if in-degree and out-degree of each vertex are equal. The Graph need not be connected.

2. **Solution Graph (S):** Given a reallocation A, the solution graph S models A as an Eulerian subgraph of request graph D. The edge from a branch to student indicates the previous branch of the student. The edge from student to branch indicates the new allotted branch to the student. If a student node doesn't have any edges, it means that the student did not change his branch in the reallocation. The edges between the Buffer and branches are to ensure that the graph is Eulerian.

3. **Partial Solution Graph (S_i):** During the run of the algorithm in section 4.3, the allocations made till a point is referred to as a Partial solution and the corresponding graph representation as described above is called as Partial Solution graph.

4. **Residual Digraph (R_i):** After updating the partial solution in each iteration, we update the request graph by reversing certain edges and removing certain others. This graph is called Residual Graph.

5. **Algebraic Addition of Directed Graphs:** Let the two graphs be G_1 and G_2. Here $G_1 + G_2$ implies the *Algebraic Addition* of the graphs G_1 and G_2 i.e $(V_1 \cup V_2, E_1 \cup E_2)$ but if there is an edge $E_{i,j}$ in G_1 and edge $E_{j,i}$ in G_2 then there will be no node edge between the vertices i, j in the resultant graph. (If the are edge between two vertices in the opposite direction they cancel each other in the resultant graph)

Let us call the digraph constructed above as the Request graph and denote it by D.

Every iteration i of the algorithm below starts with a *partial solution* S_{i-1} and a *residual digraph* R_{i-1}, and constructs a new partial solution S_i and a new residual graph R_i. S_0 is initialized as the empty digraph on V, and R_0 is initialized as D.

Observations:

1. Every partial solution S_i is an *Eulerian digraph* which is a subdigraph of R_0 which is same as D.

2. After every iteration, the residual graph can be considered as a request graph in which the number of students is one less than the previous.

4.3 Algorithm

1. Sort the students in descending order of their GPAs and store it. If the GPAs are equal we consider the JEE rank as tie-breaker.

2. Now consider the graph we constructed above and select the student with the highest GPA. Select his first preference and find a directed cycle in R_0 containing this arc, if directed cycle is not found select second preference and so on. Let us call this cycle as C_1. This is also the partial solution S_1. If no cycle is found for any of his preferences then it would be an empty cycle i.e he will remain in the same branch.

3. If a directed cycle C_1 is found in the previous step reverse all the edges in the R_0 that are present in cycle C_1 and then remove the current student and all the edges incident on him.

4. Now in this residual graph R_1, we go to the next highest GPA student and find a cycle (C_2) in the order of his preference.

5. Now we algebraically add partial solution S_1 and the cycle C_2 found in the previous step. The resultant is the partial solution after two iterations S_2.

6. Now we again construct the residual graph R_2 using R_1 and C_2 as we did in step-3. Repeat this process for all the students in the order of GPAs.

7. While exploring a student it may happen that in the residual graph the student node is incident on has reversed arcs because we are reversing the arcs in the previous steps while constructing residual graph.

8. While trying to find a cycle containing an arc from the student under consideration to his preference under consideration if the arc is already reversed then the partial solution S_i will be same as S_{i-1} and the residual graph R_i will be the graph after removing the student and all the arcs of current student from R_{i-1}.

9. After s iterations the last partial solution i.e S_s will be returned as the final solution for the given set of requests for branch change. Where s is the number of students applied for branch change.

4.4 Proof of Correctness

In this Section we show that the Algorithm described in the previous section gives the best solution as per supremum norm[2.1.5].

4.4.1 Notations and Definitions

1. **Request Graph (RG_i) :**This is the graph constructed for $(i+1)$-th iteration.

2. **Optimal Solution** $OS(RG_i)$ **:** This the best possible solution for a given set of requests and branch strengths which is represented as a graph (may be found from the brute force method).

   $OS(RG_i)$ is the graph constructed from the optimal solution by adding the following edges: 1)From student to his alloted branch. 2) From current branch of student to student. Then adding the edges from buffer to branches and from branches to buffer to make the graph eulerian, i.e adding the edges from branch to buffer if the number of incoming edges to branch is more than outgoing edges and viceversa.

   Optimality: $\{\ s_i \rightarrow b_i, s_{1+1} \rightarrow b_{i+1}, ..., s_n \rightarrow b_n,\} \in OS(RG_i)$

   where ($s_j \rightarrow b_j$) implies student s_j has been alloted a branch b_j .

3. From the construction of residual graph (request graph for next iteration) we can see that

$$RG_1 = (RG_0 - 2C_1) \setminus \{s_1\}$$

$$RG_2 = RG_1 - 2C_2 \setminus \{s_1, s_2\}$$

$$RG_2 = RG_0 - 2(C_1 + C_2) \setminus \{s_1, s_2\}$$

$$RG_2 = RG_0 - 2S_2 \setminus \{s_1, s_2\}$$

$$\mathbf{RG_i = RG_0 - 2S_i \setminus \{s_1, s_2, ..., s_i\}}$$

Claim: The partial solutions S_i assigns $\{\ s_1 \rightarrow b_1, s_2 \rightarrow b_2, ..., s_i \rightarrow b_i,\}$

Base Case: To show that the claim holds for $i = 1$.

Claim:$\mathbf{OS(RG_0)} \subseteq \mathbf{RG_0}$.

This is true because in $OS(RG_0)$ there are edges from current branch to student and student to alloted branch which all of these are present in the reguest graph RG_0. The edges that are added because of buffer are also present in the request graph because if there are unbalanced edges for the branches in $OS(RG_0)$ it means that the branch strength may increase or decrease but as this is an optimal allocation the edges would be less than or equal to the no.of edges in the request graph.

Now as $OS(RG_0) \subseteq RG_0$ and $(s_1 \rightarrow b_1)$ is present in $OS(RG_0)$,There would be atleast one cycle C_1 which includes $(s_1 \rightarrow b_1)$. So in our algorithm will find a cycle for sure which includes this edge.

As $C_1 = S_1$, The claim holds true for $i = 1$.

Inductive Step: To show that for any $i \geq 1$, if the claim holds for S_i then the claim also holds for S_{i+1} .

Assume that the claim is true for S_i.

Now we are in the $(i+1)$-th iteration i.e we have RG_i and S_i .

Two cases are possible here

1) The edge $(s_{i+1}, b_{i+1}) \in E(RG_i)$

2) The edge $(s_{i+1}, b_{i+1}) \notin E(RG_i)$

Where $E(RG_i)$ is the set of edges of RG_i.

Suppose b is a branch which the student s_{i+1} prefers over b_{i+1} which is what the optimal solution assigns to her. Notice that RG_i does not have any cycle C containing (s_{i+1}, b). Otherwise $C + S_i$ will be a solution that is better than the optimal solution under the supremum norm.

Case:1 The edge $(s_{i+1}, b_{i+1}) \in E(RG_i)$

$$OS(RG_0) \subseteq RG_0$$

$$OS(RG_0) - S_i \setminus \{s_1, s_2, ..., s_i\} \subseteq RG_0 - 2S_i \setminus \{s_1, s_2, ..., s_i\}$$

Since $OS(RG_0) \subseteq RG_0$, we only need to consider those edges which are in S_i. If an edge is both in $OS(RG_0)$ and RG_0 and present in S_i, then we are deleting in the first and reversing it in the second. And if an edge is not present in $OS(RG_0)$ and is present in RG_0 and in S_i then we are adding the reverse in both the graphs. In both the cases the first is still a subgraph of the second. Hence this holds.

$$RG_i = RG_0 - 2S_i \setminus \{s_1, s_2, ..., s_i\}$$

$$(RG_0 - 2S_i) \setminus \{s_1, s_2, ..., s_i\} = RG_i$$

$$(OS(RG_0) - S_i) \setminus \{s_1, s_2, ..., s_i\} \subseteq RG_i$$

There exists a cycle C_{i+1} containing $(s_{i+1} \rightarrow b_{i+1})$

$$C_{i+1} \subseteq RG_i$$

This is true because $(s_{i+1} \rightarrow b_{i+1})$ is part of the cycle C_{i+1} in $OS(RG_0) - S_i$.

All the vertices of $s_1, s_2, ..., s_i$ are isolated in $OS(RG_0) - S_i$. Therefore $(OS(RG_0) - S_i) \setminus \{s_1, s_2, ..., s_i\}$ is Eulerian. And also as $(s_{i+1} \rightarrow b_{i+1}) \notin S_i$ there will be a cycle involving this edge in $OS(RG_0) - S_i$.

Hence $C_{i+1} \subseteq OS(RG_0) - S_i \setminus \{s_1, s_2, ..., s_i\}$.

Case:2 The edge $(s_{i+1}, b_{i+1}) \notin E(RG_i)$

If this is the case then $(s_{i+1}, b_{i+1}) \in S_i$.

This is true because in the algorithm we remove an edge only if it is a part of a cycle in a iteration. We know that this edge is present in RG_0, but not present in RG_i which means that this edge is part of some cycle and is included in S_i.

Therefore in the current iteration i.e $(i+1)th$ the cycle $C_{i+1} = \emptyset$

$$C_{i+1} = \emptyset$$

$$S_{i+1} = S_i$$

From the above two cases we can see that in any case that the edge (s_{i+1}, b_{i+1}) will be included in S_{i+1} at the end of $(i+1)th$ iteration, which shows that the claim holds for S_{i+1}.

Since both the base case and the inductive step have been performed, by mathematical induction the claim holds for all i.

Chapter 5

Implementation and Results

5.1 Implementation

- The Programming Language used in this project is C++.
- The algorithm is implemented mostly using the Boost Graph Library(BGL) which supports a lot of Graph models, functionalities and algorithms.
- Adjacency List representation of the graph is used in modeling the graph and Dijkstra's shortest path algorithm is used to find the cycle in every iteration.
- The Graphs are also visualised using Graph Description Language DOT(Xdot format).
- The Source code is available in the following git-repository [15] .

5.2 Results

We ran this algorithm on the actual branch change requests of 2018 batch which has 24 students.The details are as follows:

Branch Details:

Branch	Sanctioned Strength	Existing Strength
CSE	50	48
EE	50	48
ME	30	28
CE	30	29

Student Details:

ID	CGPA	Current Branch	Pref1	Pref2	Pref3
1	9.318	EE	CSE	-	-
2	9.182	EE	CSE	-	-
3	9.091	CE	ME	CSE	EE
4	9.045	EE	CSE	-	-
5	9.045	EE	CSE	-	-
6	8.955	EE	CSE	-	-
7	8.909	ME	CSE	EE	-
8	8.727	EE	CSE	-	-
9	8.545	CE	ME	EE	CSE
10	8.545	CE	EE	ME	-
11	8.5	ME	EE	-	-
12	8.364	CE	CSE	EE	-
13	8.227	CE	EE	-	-
14	8.182	ME	EE	-	-
15	8	CE	CSE	EE	ME
16	7.727	ME	EE	-	-
17	7	ME	EE	-	-
18	6.455	EE	CE	-	-
19	5.909	EE	CE	-	-
20	8.273	EE	CSE	-	-
21	8.227	EE	CSE	-	-
22	7.818	CE	ME	-	-
23	7.591	CE	ME	-	-
24	6.864	CE	CSE	ME	-

The Request Graph before starting the algorithm is shown in 5.1

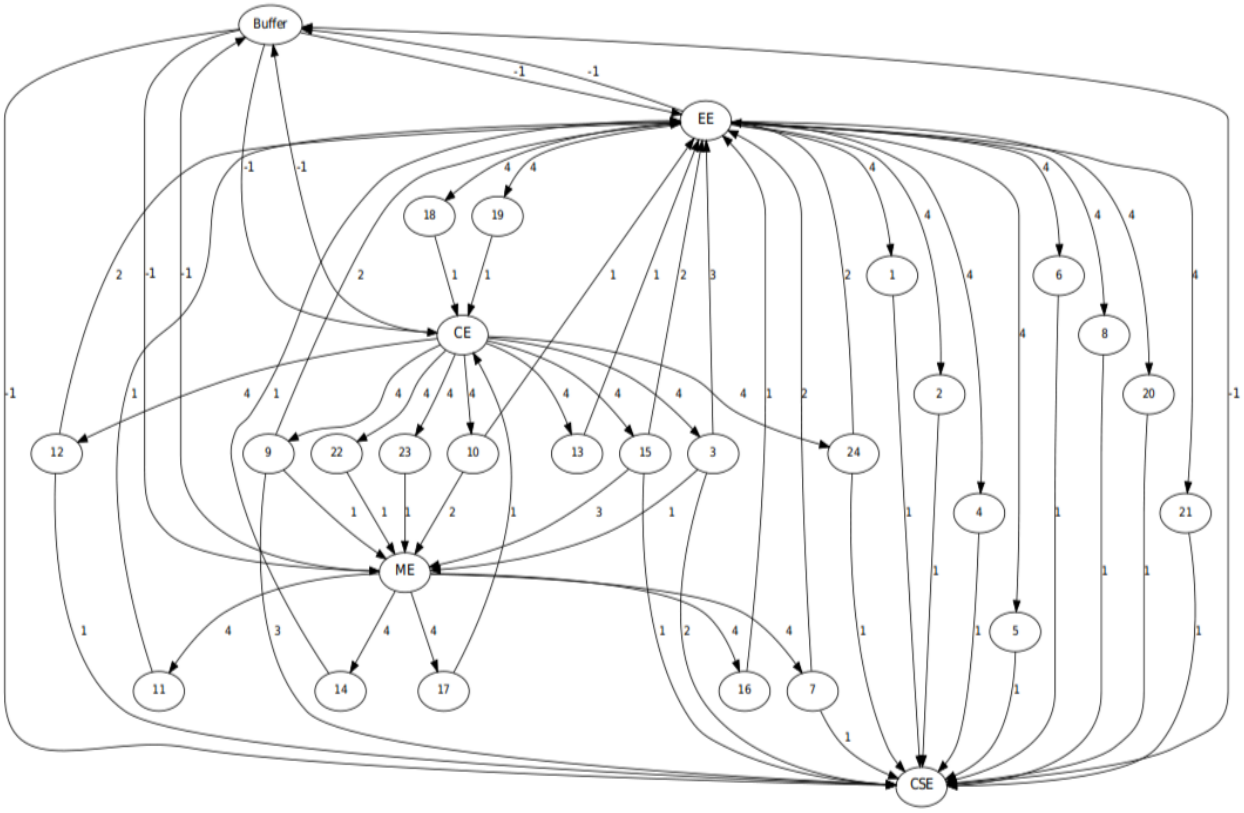

Fig. 5.1 Request Graph

The edge weights in the graph indicate the priority of a student. The edge weights for buffer is given as -1.

The Solution Graph after running the algorithm is shown in Fig-2.

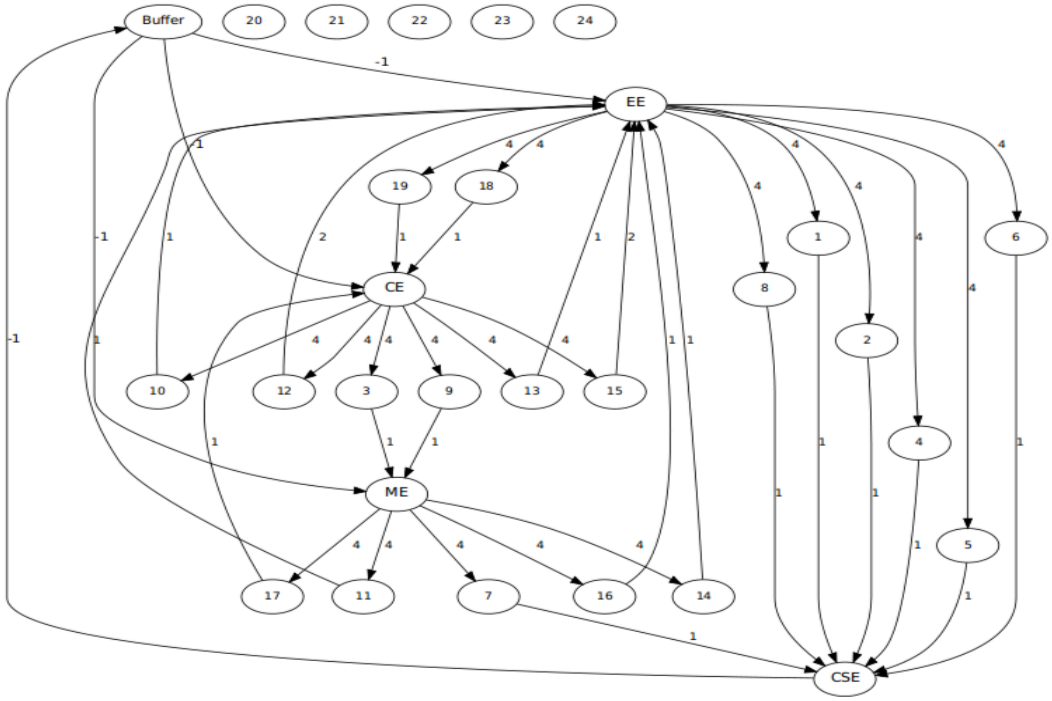

Fig-2 : Solution Graph with priorities.

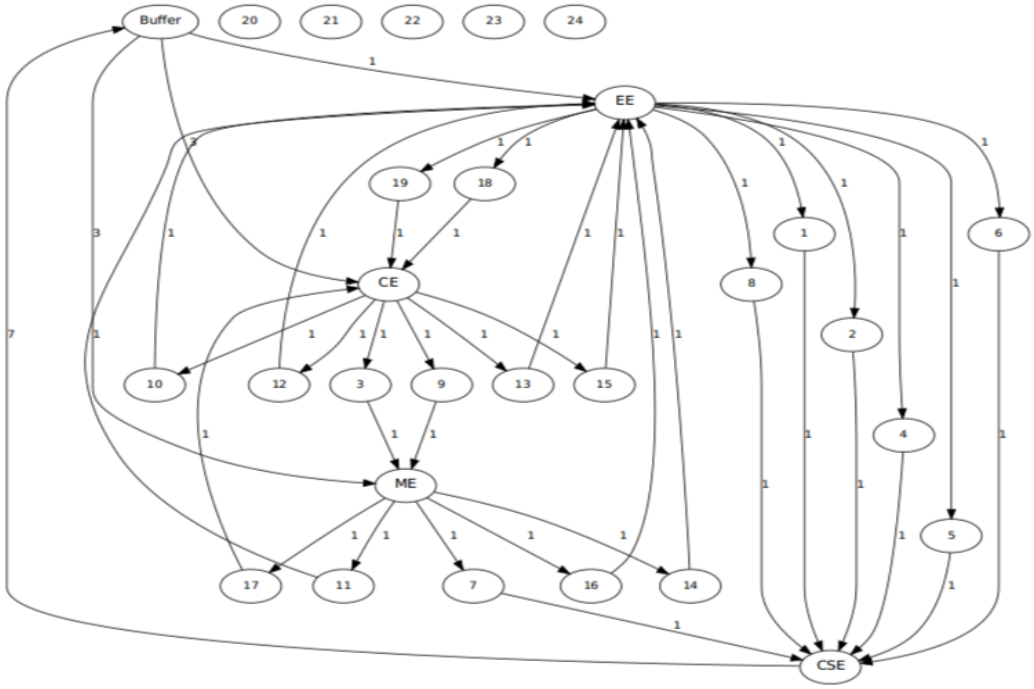

Fig-3 : Solution Graph with count (multiple edges).

Understanding the Solution Graph

- Out of 24 students who applied for branch change 19 students changed the branch and 5 students remain in their same branch.
- The in-edge for a student vertex indicate his original branch and the out-edge indicate his new allocated branch.
- The edge weights indicate the student's priority. In this allocation 17 students got their 1st preference and 2 students got their 2nd preference.
- For every Branch vertex, an out-edge to a student indicate that the student left the branch and an in-edge indicates that a student joined the branch.
- For the Buffer vertex, an in-edge from a branch indicates the no.of students added to that branch. In Fig-3, there is an edge from CSE to buffer with weight of 7, it means 7 students are added to CSE branch in this allocation.
- For every out-edge for a buffer vertex to a branch indicate the no.of students the left the branch. In Fig-3, there is an edge from Buffer to CE with weight 3, it mean that 3 students left CE in this allocation.
- In Fig-3, We can see that for every vertex, the in degree and out degree are equal. Ex:Buffer:In-edges: 7 edges from CSE, Out-edges: 1 edge to EE, 3 edges to ME, 3 edges to CE.

This allocation exactly matches with the allocation that the academic section has arrived at for this set of branch change requests.

This algorithm takes less than a second for execution with 4 branches and 20 students, where as the brute force algorithm takes around 58 minutes to arrive at the solution.

The time complexity of this algorithm is $O(s^2b^2log(s+b)$ where s is the number of students applied for branch change and b is the number of branches.

5.3 User Interface

We developed a web interface using Django web framework where the user can upload a csv file which contains the branch change requests and the branch strengths and get the optimal allocation for the given set of requests in a csv file which contains student, original branch and the new branch.

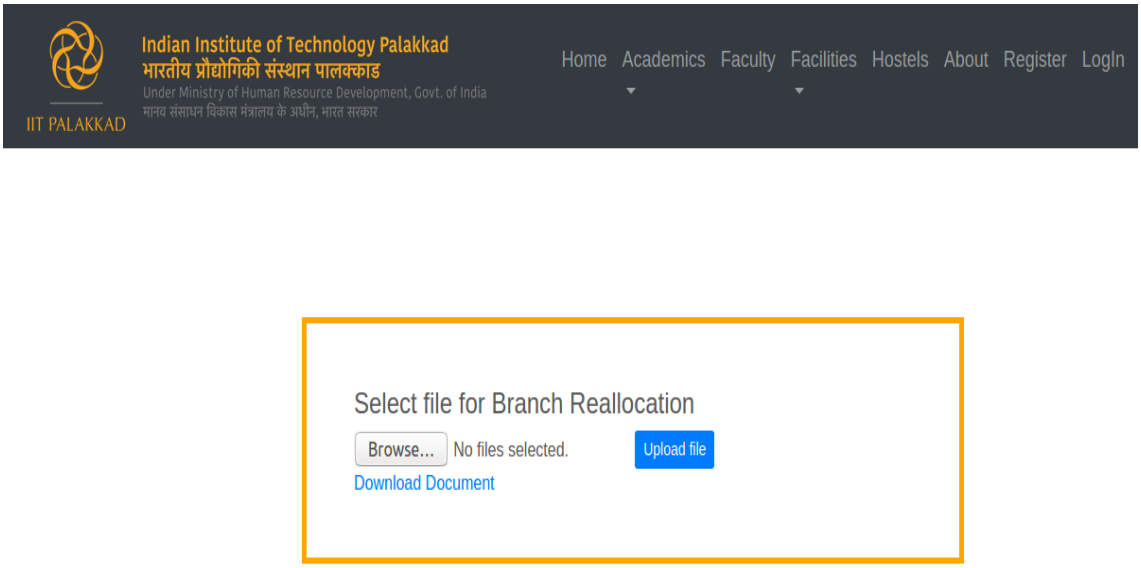

Fig-4 : A snapshot of the web interface.

Chapter 6

Difficulty with Rule - 5

Rule-5: If a student with a higher GPA is not offered a particular branch because of other constraints, this should not be offered to any other students with a lower GPA even if he/she is eligible on the basis of the existing norms.

The immediate solution that can be thought of is, in our algorithm without Rule-5, we can block incoming to a branch by marking the branch as restricted once a student is rejected for a that branch. But this doesn't solve our problem in all the cases. For instance this modification doesn't work when more than one branch is bottomed out. See 6.1.

The following are the attempts we made to find a solution for the sub-problem i.e the case where all the branches are **Bottomed-out** (i.e No Buffer is available and cyclic Swaps of students between the branches are the only possibilities). We haven't been able to find a solution so far. We suspect that inclusion of Rule-5 makes the problem NP-Hard.

6.1 Algorithm 1

1. Construct the Graph as in 4.1.

2. Sort the students in descending order of their GPAs and store it.

3. Every iteration starts with a Request Graph(R_{i-1})and a Partial Solution (S_{i-1}).

4. In every iteration we select the student with Highest GPA in the Request Graph (R_{i-1}) and try to find a path from the student's requested branch to current branch. But we try to find this path in the reverse order starting with student's current branch and following the arcs in the reverse direction using **Reverse_Path_Finder** function.

5. Inputs for Reverse_Path_Finder Function are Request Graph(R_{i-1}), Current Branch and Requested branch of the Student. The expected output is a cycle involving the arc from current student to current preference under consideration.

6. After every iteration we algebraically add the cycle returned by Reverse_Path_finder to the existing Partial Solution.

7. If a directed cycle is found in the previous step reverse all the edges in the R_{i-1} that are present in cycle C_i and then remove the current student and all the edges incident on him. This will our new request graph R_i.

8. After s iterations the last partial solution i.e S_s will be returned.

6.1.1 Reverse_path_finder

1. We first reverse all the edges in the Request Graph and remove the vertex of current student, let us call this graph as (RR_i).

2. We start with student's current branch as our start vertex and set goal vertex as student's requested branch.

3. In the process of finding the goal vertex, if we are at any student vertex, there will be only one possible next vertex i.e student's current branch in R_i. We set this as our next vertex and add it to our path. Then we reverse this edge (i.e edge between student and next vertex) in RR_i

4. If we are at any branch vertex, we find the student with highest GPA among all the out-edges for this branch irrespective of the preference number. (Otherwise Rule-5

may be violated.) We set that student as next vertex and add it to our path. Then we reverse this edge (i.e edge between branch and highest GPA student requesting this branch) in RR_i.

5. Once we reach the goal vertex, we add the original student and add current and requested branch edges and then reverse this complete cycle and return it.

The idea of reversing every edge while traversing in Reverse_Path_Finder in step-3 and step-4 can be helpful in the cases where the optimal solution graph is not a connected graph (or is a union of Eulerian Cycles). This can be seen in Example-1.

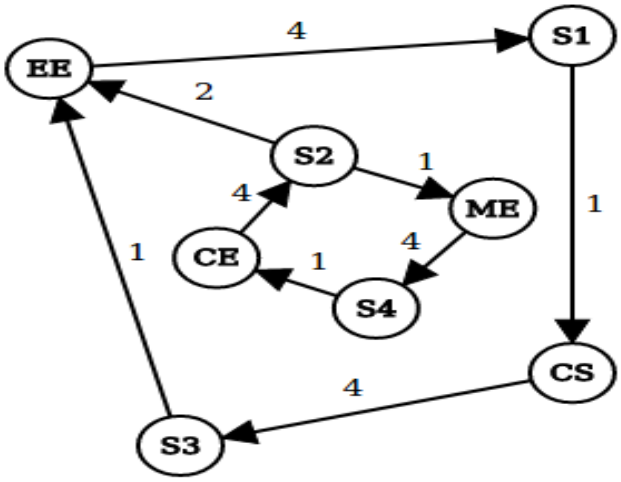

Fig. 6.1 The Reverse_Path_Finder starts with the path S1, EE, S2, CE, S4, ME, S2. Since EE to S2 arc is RR_{i-1} is reversed, this path can continue to EE, S3 and finally reach our goal CS.

6.1.2 Observations

1. This algorithm works fine for the case where every student gives atmost one preference. But it may fail otherwise.

2. This algorithm fails when the preferences of $S2$ in 6.1 are interchanged. See 6.2. Then

the solution from Algorithm 1 will not respect rule-5 because $S2$ will not get his 1st preference EE,where as $S3$ a student with GPA less than $S2$ will get EE.

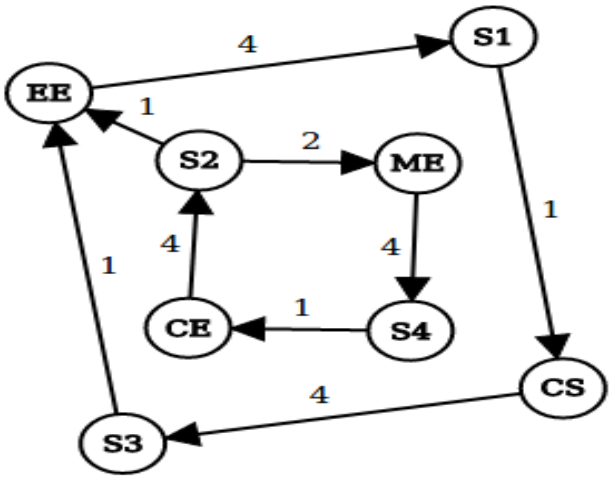

Fig. 6.2 The Reverse_Path_Finder returns a cycle which is union of S1, CS, S3, EE, S1 and S2, ME, S4, CE.This solution violated Rule-5 since S3 got EE but S2 did not even though it was his first preference.

3. There is also a possibility that the algorithm might reverse the older allocations in the same iteration, because a branch vertex can be visited any number of times. This can be seen in 6.3.

We tried certain heuristics to solve the difficulties mentioned above. For example, maintaining how many times a student vertex can be visited , classifying which edges or preferences are live and dead. But none of them was sufficient to guarantee an optimal solution. Since the above algorithm did not solve our problem we designed another algorithm.

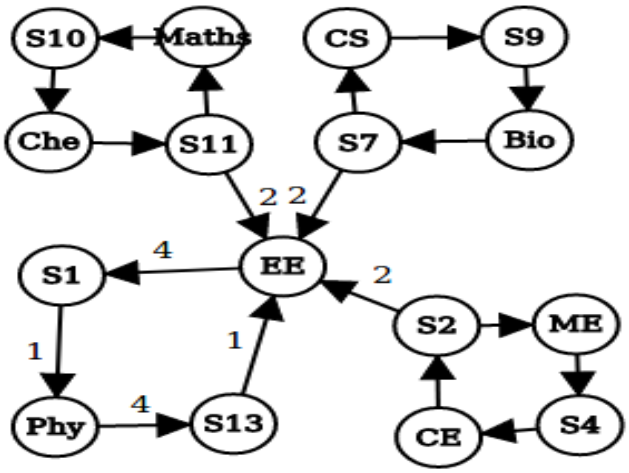

Fig. 6.3 Here in the iteration for S1, we complete the inner cycle of S2 (all the edges will be reversed) and then proceed to S7, but after completion of S7 cycle, the algorithm will again pick S2 and reverse all the edges in the cycle again.

6.2 Algorithm 2

1. Construct the Graph as in 4.1.

2. Sort the students in descending order of their GPAs and store it.

3. Select the student with the Highest GPA and find a reverse path from his current branch to his requested branch using **Reverse_Path_Recursion Function**.

4. Inputs for Reverse_Path_Recursion Function are Request Graph(R_{i-1}), Current Branch and Requested branch of the Student. The expected output is a cycle involving the arc from current student to current preference under consideration.

5. After every iteration we algebraically add the cycle to the existing Partial Solution.

6. After s iterations the last partial solution i.e S_s will returned.

6.2.1 Reverse_path_Recursion

1. We first reverse all the edges in the Request Graph and remove the vertex of current student, let us call this graph as (RR_i).

2. We start with Student's current branch as our start vertex and set goal vertex as student's requested branch.

3. In the process of finding the goal vertex, if we are at any student vertex, there will be only one possible next vertex i.e student's current branch in R_i. We set this as our next vertex and add it to our path.

4. If we are at a Branch vertex, we find the student with highest GPA requesting for this branch. If this student's preference is 1, then we set this student as next vertex and add it to our path.

5. If the preference is not 1, then we recursively call the Reverse_Path_Recursion function for this student and try to fulfill his 1st preference, so that he will not be a competitor for this branch and next Highest GPA student requesting this branch will can be taken.

6. We algebraically add the solutions returned by each recursive call.

7. Once we reach the goal vertex, we add the original student and add current and requested branch edges and then reverse this complete cycle and return it.

6.2.2 Observations

1. In this algorithm, one may have noticed that no edge is being reversed either during the run or in the next request graph. In fact we can delete the students in the request graph who are already present in the partial solution. This is true because at every branch node we are picking the top student requesting that branch as next node while

finding the path, where as in 4.3 we are picking any possible path and keep updating it in the next iterations.

2. The solution from this algorithm respects Rule-5, But this doesn't respect the supremum norm. Counter Example is shown in 6.4.

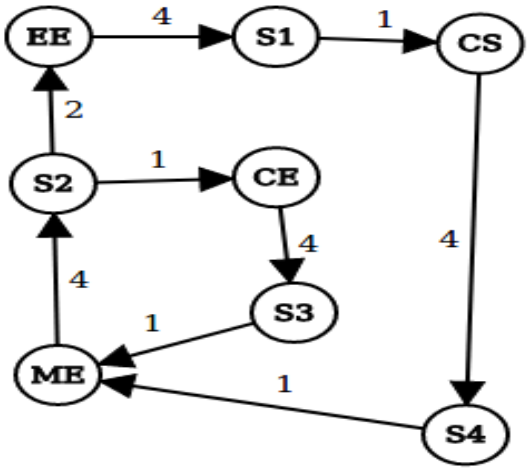

Fig. 6.4 Here the algorithm output would be $(S1 \rightarrow EE, S2 \rightarrow CE, S3 \rightarrow ME, S4 \rightarrow CS)$. But the optimal allocation is $(S1 \rightarrow CS, S2 \rightarrow EE, S3 \rightarrow CE, S4 \rightarrow ME)$.

3. We tried a backtracking approach to solve the above issue. But still the algorithm did not give the optimal solution.

Since our attempts of finding a polynomial time algorithm failed, we also tried to check whether this problem is NP-Hard by reducing known NP-Hard problems to it. We tried to reduce the problems in [3], [4] and [5] to our problem. But we were not successful in finding a reduction.

Chapter 7

Conclusion and Future Work

We have successfully designed an polynomial time algorithm for the reallocation problem with supremum rule. Time complexity of our algorithm is ($O(s^2b^2log(s+b))$ compared to exponential time ($O(b^s)$) of the Brute force solution for the problem with only supremum rule. Where s is the number of students and b is the number of branches.

We believe the Reallocation problem with both supremum criteria and Rule-5 is NP-Hard but we are yet to prove it.

References

[1] Kasireddy Durga Prasad Reddy, IIT Palakkad. **BTech Project Report, 2018**.

[2] Ordinances and Regulations **IIT Palakkad**,
`https://iitpkd.ac.in/sites/default/files/2018-01/Ordinances_Regulations_0.pdf`

[3] Jeffrey M. Jaffe. *Algorithms for Finding Paths with Multiple Constraints* , IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598

[4] Orna Kupferman and Gal Vardi. *Eulerian Paths with Regular Constraints* , School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel

[5] Marek Cygan. Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, Ildikó Schlotter *Parameterized Complexity of Eulerian Deletion Problems.*

[6] Ordinances and Regulations IITK

[7] Ordinances and Regulations IITK

[8] Ordinances and Regulations IITM

[9] Ordinances and Regulations IITB

[10] Ordinances and Regulations IITG

[11] Ordinances and Regulations IITR

[12] Ordinances and Regulations IITBBS

[13] Ordinances and Regulations IITJ

[14] Ordinances and Regulations IITT

[15] Source Code of the Project

`https://github.com/VishnuTeja27/BTP-BranchChange`